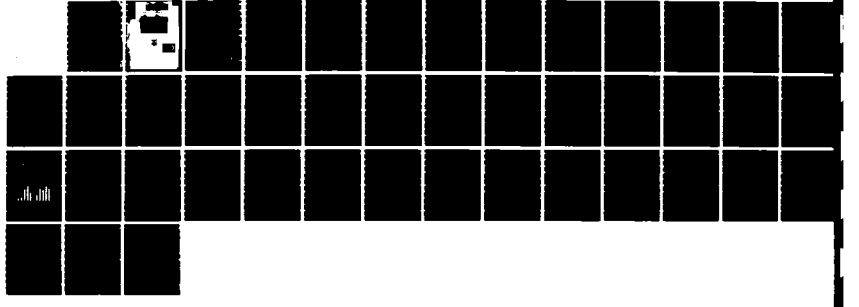
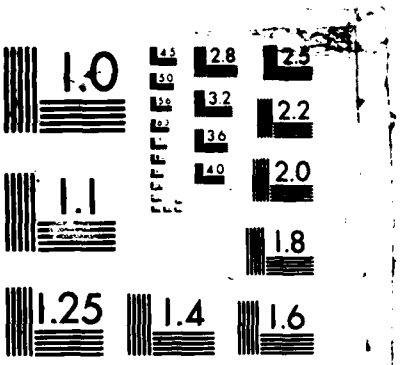
AD-A179 373 EXPERT PROGRAMMER COMPREHENSION OF COMPUTER PROGRAMS 1/1
(U) CHICAGO UNIV ILL GRADUATE SCHOOL OF BUSINESS
N PENNINGTON 01 DEC 86 N00014-82-K-0759

UNCLASSIFIED F/G 9/2 NL

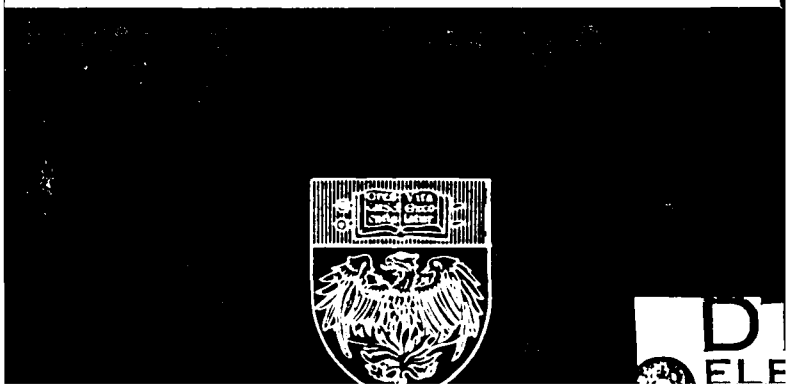




Expert Programmer Comprehension
of Computer Programs:

Final Report

Nancy Pennington
Graduate School of Business
Center for Decision Research
University of Chicago



12

Expert Programmer Comprehension of Computer Programs:

Final Report

Nancy Pennington
Graduate School of Business
Center for Decision Research
University of Chicago

December 1, 1986

This research was sponsored by the Personnel and Training Research Programs, Psychological Sciences Division, Office of Naval Research, under Contract No. N00014-82-K-0759, Contract Authority Identification No. NR667-503.

unclassified
SECURITY CLASSIFICATION OF THIS PAGE

ADA179373

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Graduate School of Business University of Chicago 6b. OFFICE SYMBOL (*If applicable*)	7a. NAME OF MONITORING ORGANIZATION Personnel and Training Research Programs Office of Naval Research (Code 1142PT)
6c. ADDRESS (*City, State, and ZIP Code*) 1101 E. 58th Street Chicago, IL 60637	7b. ADDRESS (*City, State, and ZIP Code*) 800 North Quincy Street Arlington, VA 22217-5000
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-82-K-0759
8c. ADDRESS (*City, State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO
61153N	RR04206	RR04206-0A	NR667-503

11. TITLE (*Include Security Classification*)
Expert Programmer Comprehension of Computer Programs: Final Report

12. PERSONAL AUTHOR(S)
Pennington, Nancy

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Final	FROM 9/82 TO 8/85	1986, December, 1	36

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	computer programming, expertise, text comprehension, cognitive skill, problem solving, software psychology
05	09		

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

This report summarizes research on experienced programmers' comprehension of computer programs carried out during the 36-month contract period of September 1, 1982 through August 31, 1985. Based on an extensive review of the programming skill literature, we proposed an analysis of programs based on multiple abstractions (points-of-view) that characterize program text and design. Research questions concerned how multiple abstractions are coordinated into effective mental representations necessary to comprehend programs; how different kinds of programming knowledge enter into program comprehension; what comprehension strategies distinguish those programmers who obtain high levels of comprehension from those who do not. Our research results suggest a two-stage model of comprehension. In the first stage, procedural representations dominate program understanding; in later stages, functional representations appear to dominate. Changes in the dominant representation were more extreme for programmers who talked out loud while working, suggesting that both time and task demands influence the nature of program understanding.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	Unclassified

22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (*Include Area Code*)	22c. OFFICE SYMBOL
Dr. Michael Shafto	202-696-4596	ONR 1142PT

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

19. (cont.)

We interpreted these results in terms of van Dijk and Kintsch (1983) who propose textbase macrostructure and situation model memory representations in comprehension. The feature that distinguished the best comprehenders from the poorest in our research was the use of cross-referencing strategies in which procedural relations in program text (textbase macrostructure) were explicitly mapped onto functional relations, expressed in the language of the real-world objects to which the program referred (situation model). The poorest comprehenders tended to use singular strategies, working either at the program text level or at the real-world domain level, but not both.

Expert Programmer Comprehension of Computer Programs:

Final Report

The psychology of computer programming is important for both practical and theoretical reasons. From a practical point of view software accounts for the major portion of the cost in the the development of computer systems (Boehm, 1973). The ability to control these costs will rely in part on improving programmer effectiveness. In addition, it has become increasingly difficult to guarantee the quality of software as the complexity, variety, and sophistication of programs increases. A high quality program needs to be not only reliable and correct but easy to use, maintain, and modify. The need for empirical research in this area is highlighted by a lack of consensus among computer professionals about which particular programming methods, tools, or language features improve productivity and the quality of programming products. These disputes can be addressed directly by research on the psychology of programming.

From a theoretical point of view, computer programming is an example of a complex cognitive skill. As such, the nature of the skill and its acquisition are of substantial interest to theoretical psychology (Anderson, 1981). A cognitive theory of computer programming skill, of necessity, includes components specific to the domain of computer programming. For example, experienced programmers must have large stores of knowledge highly specific to computer programming tasks. However, computer programming skill also includes more general components such as problem solving, learning, and memory, which are common to all expert skill tasks. Thus, existing work in cognition and expert skill can help to illuminate the nature of programming skill while

research on the psychology of programming can help to extend and refine general cognitive theories (Anderson, 1983).

Review

In an early paper (Pennington, 1982), we reviewed the research on computer programming skill involving experienced programmers (excluding studies of learning to program). We divided studies into those that investigated particular "programming practices" and those that sought to develop a comprehensive theory of programming skill. The bulk of the research (over 80%) on programming fell into the first category, addressing questions concerning what features of programming practices and programming languages simplify or enhance the composition, comprehension, debugging, and modification of computer programs.

The most frequently studied programming practices have included: techniques of commenting and documenting programs, mnemonic variable naming and style of program layout; the benefits of certain language features, degrees of control flow structuring, data typing, global and local variables, and looping constructs; and the utility of particular forms of program representation such as flowcharts, program design languages, and others. The results of these dozens of studies (see Pennington, 1982, Table 1 for a summary) yield few general conclusions as to the utility of various programming practices and forms.

In our review, we argued that failures to place research on programming practices in a larger context informed by psychological theory and to analyze carefully the programming task domain have contributed to a proliferation of empirical results that are difficult to interpret. For example, many of these

studies concentrated on whether rather than how aids such as flowcharts and program design languages help in program design and comprehension. When viewed in this way (asking how), higher level issues that emerged from these studies concern what kinds of information are embedded in the program text; which kinds need to be accessed; and which kinds require an alternate representation because they are too difficult to abstract easily by ordinary processing (Green, 1980; Green, Sime, & Fitter, 1980). This requires an analysis of programming tasks and a theory of programming skill that specifies which meanings in programs are relatively clear (e.g., inferred automatically), and which information is relatively difficult to extract from program text and is critical to comprehension or other tasks.

The second, smaller body of research on programming has sought to develop comprehensive cognitive theories of programming skill (see Pennington, 1982, Table 2). In contrast to the heavily empirical flavor of the programming practices research (with some exceptions) the theories of programming skill have not been submitted to extensive empirical test (also with some exceptions). In this literature the study of programming skill has drawn on a variety of other cognitive domains such as text comprehension, planning, and other expert skill domains. Research treating programs-as-texts has focused on comprehension strategies and memory. Treatments of programming-as-planning has focused on problem solving strategies and problem decompositions. Treatments of programming-as-expert-skill has focused on the organization of knowledge specific to the programming domain that is implicated in both comprehension and construction of programs. The central questions that emerge from a review of this literature reflect questions that also emerge from programming practices

studies: a) What are the successive representations of the program as the external problem domain is transformed into a representation in the programming language? b) How do successive transformations retain or obscure information about data structure, data flow, control flow and function? c) Are there fundamental structural components that are psychologically meaningful? d) What is the nature of programming knowledge and how does it influence the execution of programming skill?

Conceptual Framework

In order to develop a framework within which to address these questions, in a subsequent paper (Pennington & Grabowski, 1985; Pennington, in press), we elaborated the idea that an essential feature of program design and program comprehension is that there are multiple interconnections between program parts that are difficult to conceptualize simultaneously (Green, 1980; Green, et al, 1980). We proposed an analysis of program designs and program texts in the form of "multiple abstractions." We use the word "abstraction" to mean a solution design in which some relations between parts are specified but others are not. For example, an architect's floorplan is one abstraction of a house plan and the exterior drawing is another. In each, certain interrelations between parts are explicitly specified, others can be inferred, and others are completely unspecified. This is also true of a rhythmic abstraction or thematic abstraction of a musical score. The utility of performing such an analysis within a problem solving domain is to identify the kinds of information "in the solution design" that need to be coordinated in the design process and detected in comprehension processes. Although these abstractions are not intended to specify mental entities, they provide a starting point for

thinking about how alternate conceptualizations of computer programs might provide a basis for mental representations at different points in program design and comprehension processes.

An example of a very simple "toy" programming problem and four different abstractions of a solution to it are presented in Figures 1 through 5. The programming problem (Figure 1) is to rearrange a table of codes so that all codes of one type are moved to the first part of a table of codes, all codes of a second type are placed in the second part of the table, plus some other marking and printing requirements.

****
Insert Figure 1 about here
****

The first abstraction of the problem solution is structured in terms of the goals of the program, that is, what the program is supposed to accomplish or produce (Figure 2). It is labeled a goal hierarchy but could also be described as a decomposition according to the major program functions or outputs. The first level decomposition shows that the program will "do" or produce three things: a rearranged table, identifying labels for each code, and some printed output. At the next level, subgoals are specified for each higher level goal. For example, rearrangement of the table involves separating "A" codes from "B" codes, putting "A" codes first into the table, and putting "B" codes second into the table. Notice that in this abstraction there is no explicit information as to how these goals will be accomplished. For example, the "A" and "B" codes could be separated and then counted up and then put back into the table. Alternatively, a single code from the table could be examined, classified as "A" or "B", added into the appropriate "A" or "B" counter, placed

PROBLEM: REARRANGE A TABLE OF CODES SO THAT ALL TYPE "A" CODES COME BEFORE ALL TYPE "B" CODES IN THE TABLE. LABEL EACH CODE AS TO ITS TYPE. PRINT OUT THE NUMBER OF "A" AND "B" CODES IN THE TABLE.

FIGURE 1. A SIMPLE PROGRAMMING PROBLEM

at the beginning or end of the table according to its "A" or "B" status, and then the next code would be examined. Some inferences about the ordering of events can be made from this abstraction on the basis of everyday knowledge, for example, a code must be classified before it can be counted as a member of the category, but details of the procedure to do this are not specified.

****
Insert Figure 2 about here
****

A second abstraction is structured in terms of processes, operating on data objects that transform the initial data objects into the outputs of the program (Figure 3). For example, Figure 3 shows that the data object "Table" is used by the process "Select A Codes" to produce a "List of A Codes" but "Table" is not itself transformed until it enters the process "Put A Codes in Table" at which point it emerges from this process as a new version of "Table." Because the flow of each data object can be traced through the series of transformations in which it participates, this is called a <u>data flow</u> abstraction. This abstraction is closely related to the goal hierarchy. For example the first level decomposition of goals is, in the goal hierarchy (Figure 2), to rearrange codes, label codes, and print. These correspond to the final data objects at the bottom of the data flow abstraction (Figure 3) which are "Table," and "Report." The goal hierarchy can be at least partly recovered from the data flow abstraction by working up from the bottom of the data flow abstraction although it requires the application of knowledge to infer the grouping of subgoals with their goals. However, in the data flow abstraction, everything that happens to a particular data object is readily available in a way that is not apparent from the goal hierarchy. In addition,

GOAL HIERARCHY: THE PROGRAM ACCOMPLISHES CERTAIN GOALS BY PRODUCING OUTPUTS. EACH LEVEL INDICATES A HIGHER ORDER GOAL IS DECOMPOSED INTO SUBGOALS.

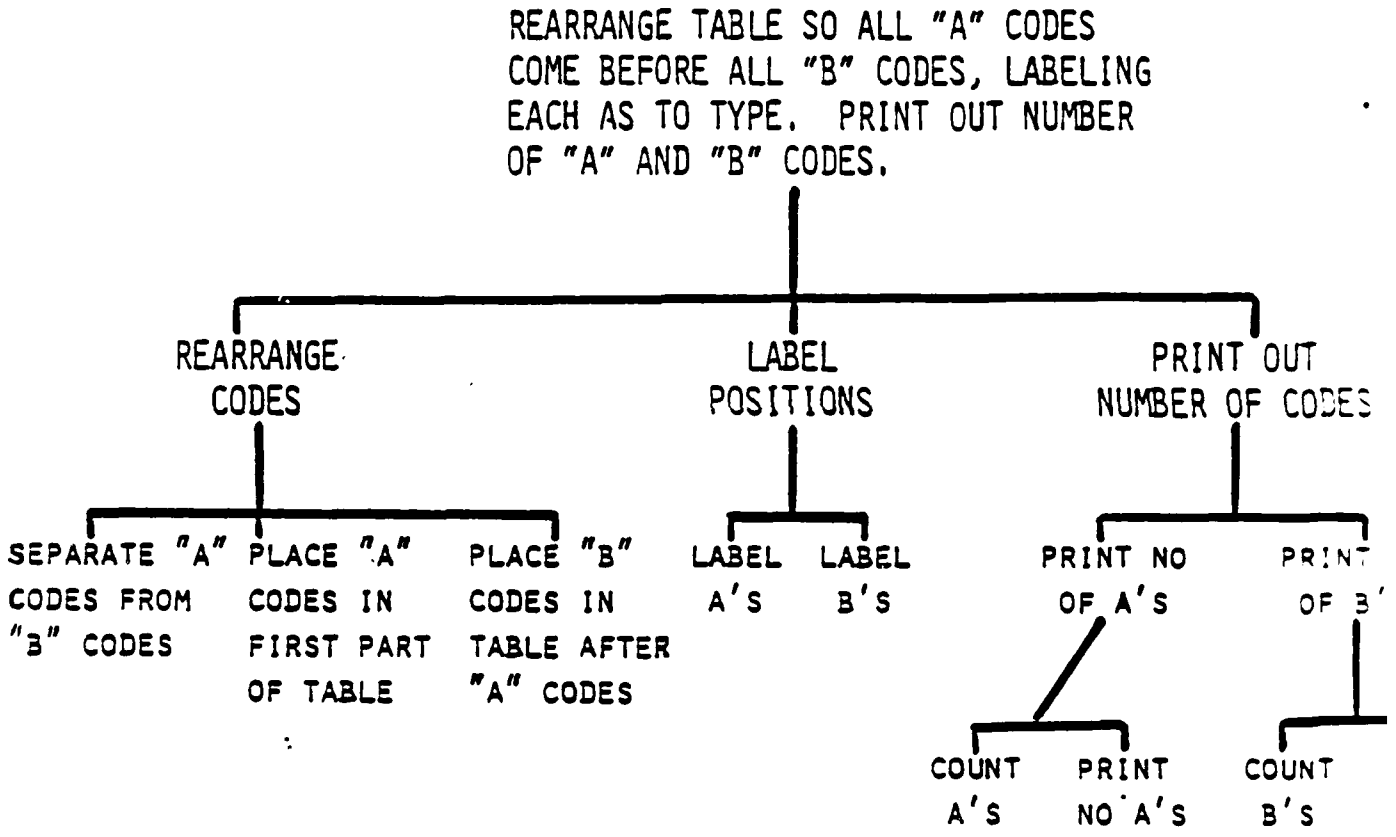


Figure 2. Abstraction of function.

this abstraction allows more inferences to be made about the order in which certain operations will occur than does the goal hierarchy abstraction. If an action (marked by a box, e.g., "Label B Positions" in the lower right of Figure 3) has a data object as an input (marked by an oval, e.g., "Table") then the action cannot take place until the data object is available; thus the process that produces a data object (e.g., "Select A Codes") must execute prior to the process that consumes it ("Count A Codes").

****
Insert Figure 3 about here
****

A third abstraction is structured in terms of the sequence in which program actions will occur (Figure 4). This is called a control flow abstraction because the links between program actions in this structure represent the passage of execution control, instead of the passage of data as in the data flow abstraction. Traditional programming flowcharts are a standard expression of a control flow abstraction. This abstraction highlights sequencing information but conclusions about data flow must be extracted by looking for repeated mentions of variable names. For example to find out in what events "Number of A Codes" participates (easily determined in the data flow abstraction, Figure 3), it is first necessary to see that this quantity is represented by a variable called "Next-A Loc" and then to track its use in the sequence of actions. To complicate things further, this variable is doing double duty as a counter of "A" codes and as a pointer to where the next "A" code goes in the table. This makes it difficult to extract goal information, even at a detailed level. So the sequence of statements involved in the subgoal "Count the Number of A Codes" (Figure 2) not only is embedded in

DATAFLOW REPRESENTATION: PROGRAM ACTIONS TRANSFORM INITIAL DATA OBJECTS INTO FINAL DATA OBJECTS. ⬭ INDICATE DATA OBJECTS. ▭ INDICATE PROGRAM ACTIONS.

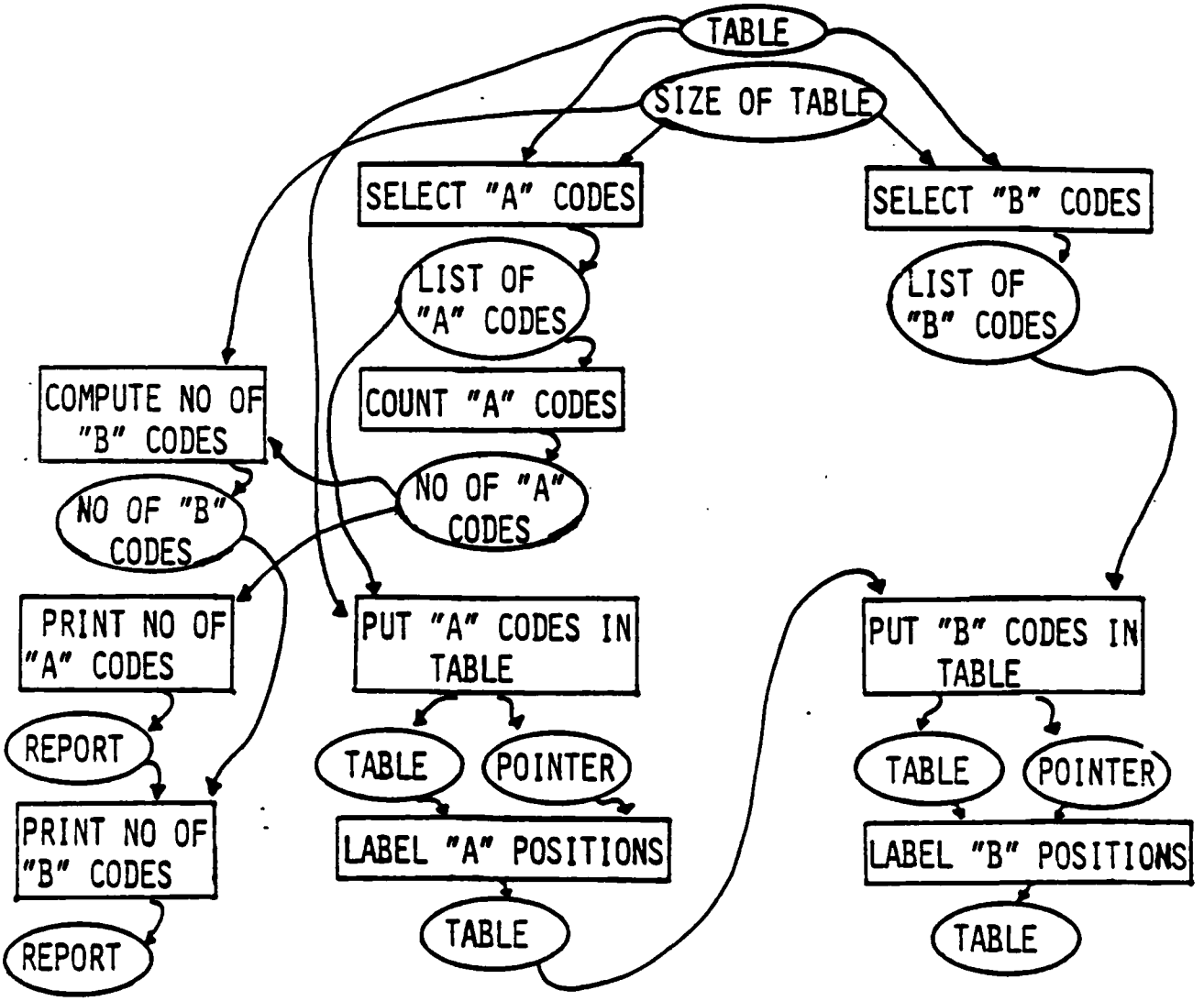


Figure 3. Abstraction of dataflow.

statements serving an indexing function, but is alo widely dispersed in the sequential abstraction (Figure 4).

****
Insert Figure 4 about here
****

A fourth abstraction is structured in terms of the program actions that will result when a particular set of conditions is true (Figure 5). This abstraction is like a decision table in which each possible state of the world is associated with its consequences; it also resembles the production system condition-action notation that is used to represent human procedural knowledge (e.g., Anderson, 1983; Newell & Simon, 1972). In a conditionalized action abstaction, the program is viewed as being in a particular state at each moment in time, that some set of conditions exists. These conditions trigger an action, execution of the action results in a new state, the new state triggers another action, and so on. In this kind of abstraction, it is easy to find out what results if a given set of conditions occurs and also relatively easy to find out what set(s) of conditions can lead to a given action. This kind of state information is much harder to deduce from the other abstractions. However, information about the sequence in which actions occur and information about higher level goals are difficult to extract in the conditionalized action abstraction (Figure 5).

****
Insert Figure 5 about here
****

This analysis of the multiple abstractions that characterize a computer program also applies to English language instructions, such as training manuals, recipes, knitting instructions, and assembly instructions written for

CONTROL FLOW REPRESENTATION: PROGRAM ACTIONS OCCUR IN A SPECIFIED SEQUENCE

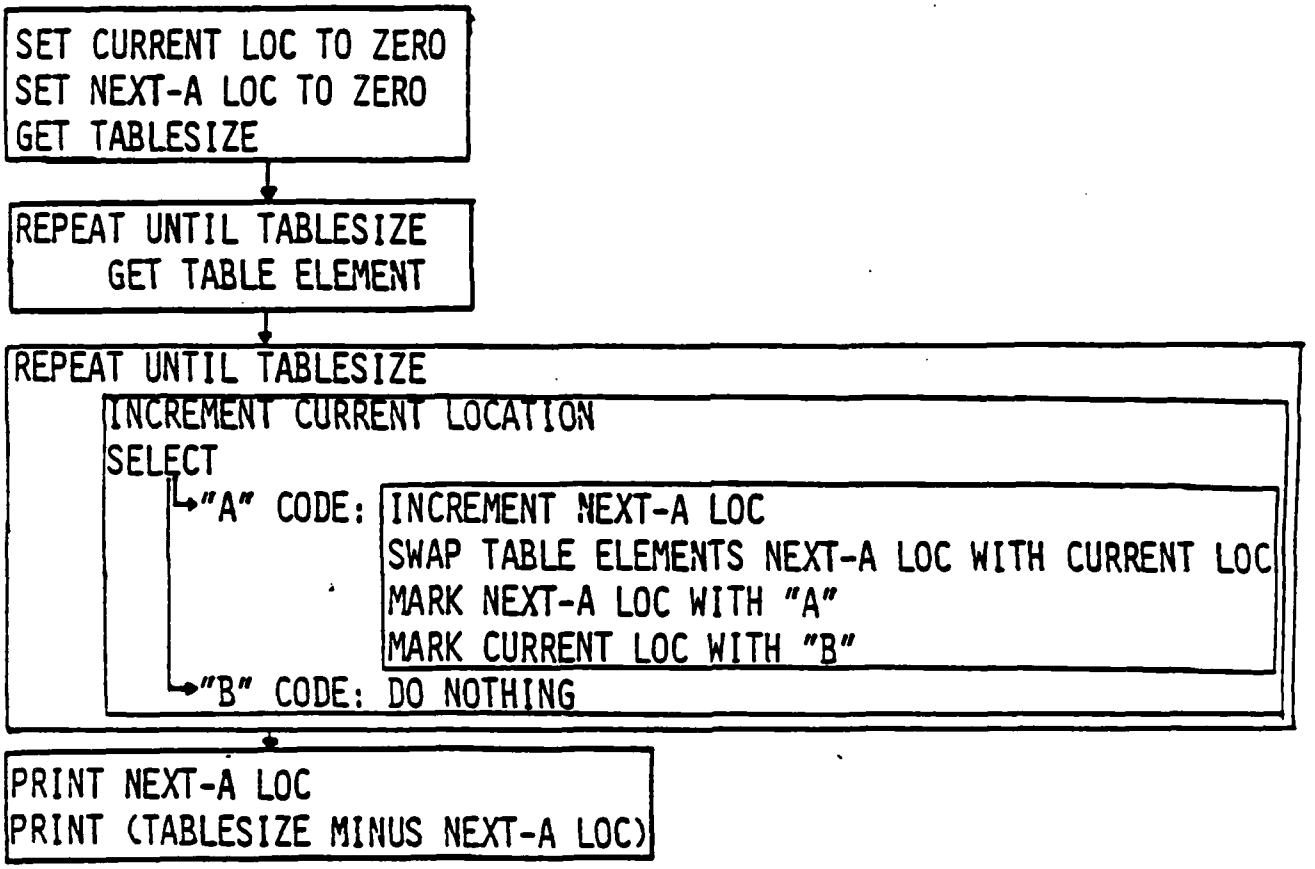


Figure 4. Abstraction of control flow.

CONDITIONALIZED ACTION REPRESENTATION: A SET OF CONDITIONS RESULTS IN THE EXECUTION OF SOME ACTION(S). THE EXECUTION OF AN ACTION RESULTS IN A NEW SET OF CONDITIONS.

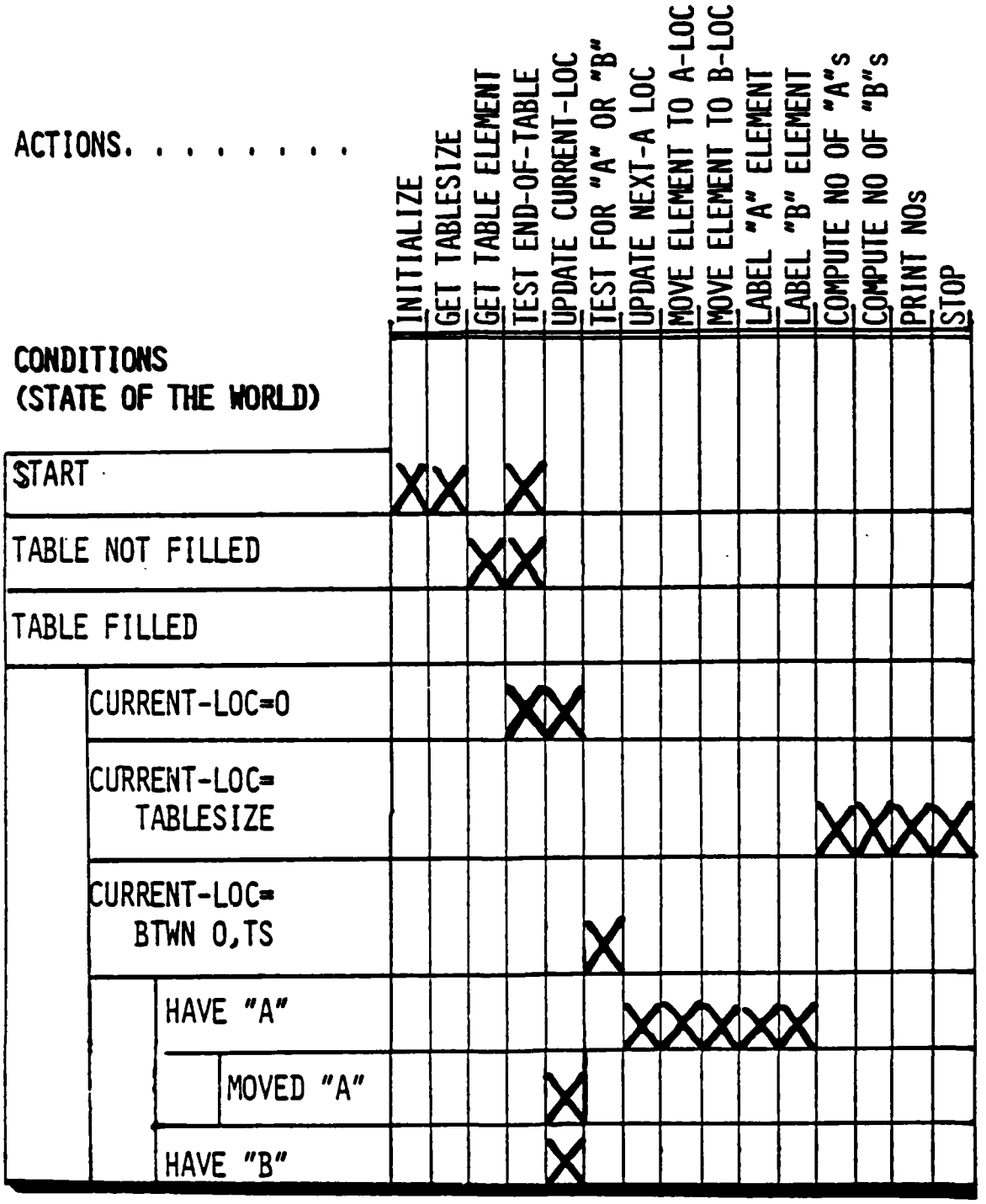


Figure 5. Abstraction of conditions and actions.

one's own future use or for another person's use. In these tasks, too, the writer wants to convey to another person what should be accomplished (goal hierarchy), how to do it (sequential procedure), the sets of conditions under which particular actions should be taken (conditionalized action), and/or the set of transformations that a particular object should go through (data flow). Part of the difficulty of writing clear instructions, and clear computer programs, is due to the tradeoffs that inevitably occur in how much of each kind of information can be highlighted simultaneously. Uncertainty about the "best" way to write instructions may be largely due to uncertainty about which of these (or other) structures should serve as the organizing principle for the instructions.

For computer programming the issue of the types of relational information necessary to describe a program is a complex one. Critical questions for programming research concern how these multiple abstractions are coordinated into effective mental representations necessary to compose or understand a program. One might also ask whether there is a psychologically dominant way of conceptualizing programs and how this interacts with programming language and programming task. Arguments about which programming languages and programming methods are easier to use and are more comprehensible may in fact be arguments about which if any of these abstractions correspond more closely with the way that programmers actually think about programs.

Research

In our conceptual framework, we argued that comprehension of computer programs involves detecting or inferring different kinds of relations between program parts. We have also argued (Pennington, in press) that different kinds

of programming knowledge will facilitate detection and representation of the different textual relations. Our first empirical research investigated the role of programming knowledge in program comprehension and the nature of mental representations of programs; specifically, whether procedural (control flow) or functional (goal hierarchy) relations dominate programmers' mental representations of programs at various stages in the comprehension process (Pennington, in press). A summary of the correspondences we proposed between textual relations (abstractions of program text), knowledge structures, and hypothesized mental representations is shown in Table 1. Features of the text activate different kinds of knowledge, some of which will provide an organizing structure for the mental representation of the text. The first two rows of Table 1 represent alternative hypotheses concerning the dominant form of the mental representations of programs.

****
Insert Table 1 about here
****

Under the first hypothesis (Table 1, row 1), knowledge of text structure plays an organizing role in the mental representation of programs during comprehension. Comprehension proceeds by segmenting statements at the detail level into phrase-like groupings that then combine into higher order groupings. Syntactic markings provide surface clues to the boundaries of these segments and the segmentation reflects the control structure of the program. Thus in terms of the multiple abstractions of programs (Figures 2 through 5), sequence information should be readily available; data flow connections that occur across unit boundaries should be relatively more difficult to infer; and function information should be least accessible since it is most closely

Table 1
Correspondences Between Text Abstractions,
Knowledge Structures, and Mental Representations

TEXT RELATIONS	KNOWLEDGE STRUCTURES	MENTAL REPRESENTATION
Control Flow	Text Structure	Procedural Episodes
Function Data Flow	Plan Knowledge	Functional Representation
Condition-Action	Unknown	Unknown

related to data flow and requires coordination across text structure units.

Under the second hypothesis (Table 1, row 2), knowledge of program plans plays an organizing role in the mental representation of programs during comprehension. Comprehension proceeds by the recognition of patterns that implement known programming plans. Plans are activated by partial pattern matches and confirming details are either sought or assumed. The resulting segmentation reflects the data flow structure of the program indexed by program function. Thus in terms of the multiple abstractions of programs (Figures 2 through 5), data flow and function information should be readily available; sequence and detail operations should be less accessible.

There are several reasons to be interested in which of these views better characterizes computer program comprehension. The nature of mental representations of programs and the units that underlie their organization (e.g., Adelson, 1984; Curtis, et al., 1984) are important for resolving arguments over how programs ought to be structured, understanding the psychological complexity of programs, and extending insight into skilled performance to an important complex task. Second, the two modes of comprehension have different consequences in terms of the kinds of information that are relatively easy or difficult to abstract from program text (Green, 1980). This in turn is important in determining standards for computer programming practices, tools, languages, and education.

The research summarized in this section (see Pennington, in press; Pennington, in preparation for full reports) was designed to operationally identify the form of mental representations of program texts, provide information about the kinds of relational information in programs that are most

accessible, and investigate the roles of two kinds of programming knowledge, text structure knowledge and plan knowledge in program comprehension.

In the first study, experienced programmers studied short program texts and responded to comprehension questions and memory tests. Short texts were used to obtain a high degree of experimental control. Although programming studies have typically used texts of this length, it is desirable to examine experimental results in more realistic settings. In the second study programmers engaged in a more natural task in which they studied a program of moderate length, made a modification to it, and responded to comprehension questions. Thus the first study provides relatively direct information concerning the form of mental representations of program text. In the second study, comprehension data provide indirect evidence concerning the same questions for a different, more natural task.

Study One

The subjects in the first study (Pennington, in press) were 80 professional programmers with an average of 10.2 years experience as professional programmers. One-half of these programmers programmed primarily in FORTRAN and one-half programmed primarily in COBOL although most knew more than one programming language and about one-half of the sample had taught at least one computer programming course.

Subjects studied 8 short program texts, answered 48 comprehension questions (6 per text), and responded to a recognition memory test for each text. For each text, an analysis was performed that designated a hypothesized memory representation under the two hypotheses shown in Table 1. The TS (text structure) analysis reflected the hypothesis that the memory macrostructure

(Kintsch & van Dijk, 1978) was organized according to procedural units in which control flow relations between program parts dominate. The PK (plan knowledge) analysis reflected the hypothesis that the memory macrostructure was organized according to functional units in which function and data flow relations between program parts dominate. Under these alternate hypotheses, different sets of program statements were proposed to be more closely related in memory.

Priming manipulations in the recognition memory tests were designed to test the alternate memory structures. Specifically, support for a TS macrostructure would be obtained if response times to targets preceded by a TS prime were reliably faster than the same targets preceded by a PK prime. If this were the case, we could infer that the items specified by the TS analysis as forming a cognitive unit were in fact "closer" in memory than were the items specified by the PK analysis. Alternatively, support for a PK macrostructure would be obtained if response times to targets preceded by a PK prime were reliably faster than the same targets preceded by a TS prime. Finally, if some response times to PK-primed targets were faster and other response times to TS-primed targets were faster, then no inferences could be drawn regarding which of the formulations more accurately portrays the nature of mental representations.

Comprehension questions were constructed to ask about different textual relations: control flow, data flow, function, and condition-action (state). Response times and error rates for different kinds of comprehension questions provided additional measures regarding relations that dominate in mental representations. Specifically, if support for a PK macrostructure were obtained with the recognition response times, then we expected to see fewer

errors and faster response times, for function and data flow comprehension questions. Alternatively, if support for a TS macrostructure were obtained with the recognition response times, then we expected to see fewer errors and faster response times for detailed operations and control flow comprehension questions.

The results of this study provided evidence that the dominant memory representation, formed during comprehension of short program texts in this experimental context, is organized by a small set of abstract program units related to the control structure of the program. More specifically, of the four program abstractions presented earlier (Figures 2 through 5), relations captured by the procedural, control flow abstraction (Figure 2) appeared to be central in comprehension in our experimental task. Furthermore, the nature of the mental unitization of these relations corresponds to the basic building blocks of sequence, iteration, and conditional identified by early advocates of structured programming.

Both recognition memory results and comprehension questions results converged to support this conclusion (Pennington, in press). In the recognition memory test, recognition occurred faster when a statement was immediately preceded by a statement in the same text structure (TS) unit than when it was immediately preceded by a statement that was not in the same text structure unit (see Figure 6). This implies that statements in the same TS unit were closer together in programmers' memory structures. This priming effect cannot be accounted for by the text surface distance between statements, by syntactic similarity between statements, or by argument repetition since these features were controlled by counter-balancing across test items.

****
Insert Figure 6 about here
****

Reponses to comprehension questions about control flow relations and program operations were answered faster and with fewer errors than were questions about data flow and function relations, supporting the idea that control flow and operation information is easier to access in memory (see Figure 7). This pattern differed for language groups and for top and bottom quartile subjects (divided according to comprehension scores on this comprehension test); COBOL programmers showed more errors on data flow questions and top quartile comprehenders were distinguished by their superior performance on function questions (see Figure 7). This suggests that the initial phases of comprehension are devoted to the comprehension of procedural relations with later phases involving function inferences.

****
Insert Figure 7 about here
****

These empirical results fit a view of program comprehension in which the meaning of program text is developed largely from the bottom up. The text is first segmented according to simple control patterns segregating sequences, loops, and conditional patterns. At this level some specific inferences are made concerning the procedural roles of the segments. Data flow and function connections often require integration of operations across separate segments. For example, calculation of an average involves an initialization, a running sum, and final calculation; these usually occur in separate procedural units. Our results suggest that these connections are made later in comprehension, and for programmers with the lowest comprehension scores they are not made

Figure 6. Study One response times for recognition memory items comparing PK-primed item times to TS-primed item times for each set of materials within language adjusted for the effects of subject group (Panel A) and for each subject group within language adjusted for the effects of materials set (Panel B).

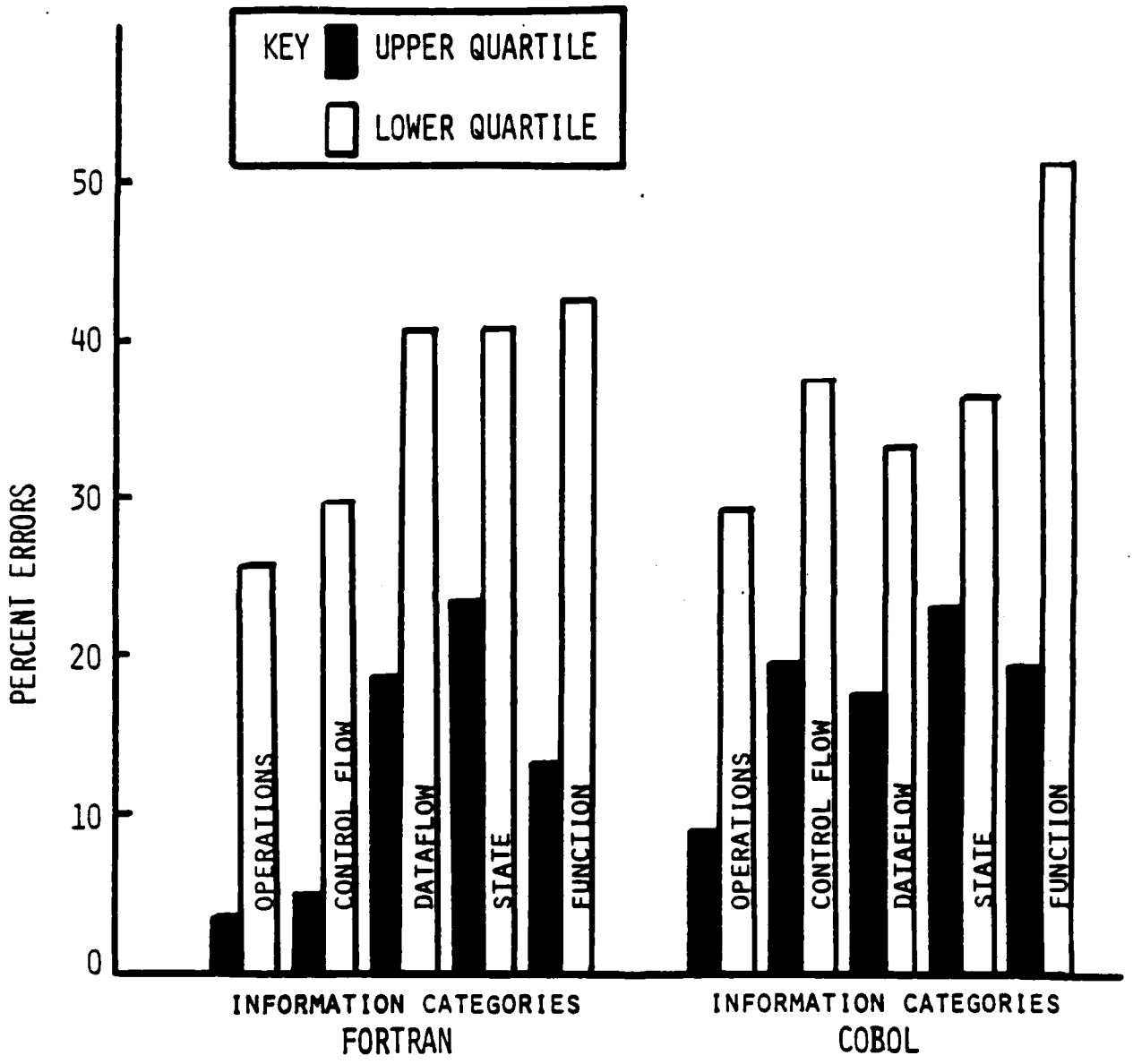


Figure 7. Study One comprehension question error rates by information category for top and bottom quartile subjects within each language.

correctly or at all within the time limits imposed by our study.

Study Two

In the first study, programmers' comprehension strategies may have been influenced by several aspects of the experimental task: short undocumented program segments, the series of short study trials, and the demands of memory questions. In a second study, a more natural programming environment was created in which programmers studied a program of moderate length (200 lines) and then made a modification to it (Pennington, in press; Pennington, in preparation). At two different points in time they were asked to summarize the program and respond to comprehension questions. Half of the programmers were asked to think aloud while they worked and the other half worked silently.

As in the first study, comprehension questions in the second study were designed to ask about particular relations between program parts: control flow, data flow, function, and condition-action relations. If the results of the previous study were to generalize to this task environment, then we expected to see good comprehension of control flow relations early in the comprehension process with comprehension of data flow and function catching up later in the process. Alternatively, data flow and function inferences would be made more readily at the outset due to the larger context in the program text used in the second study.

Forty of the 80 professional programmers who participated in the previous study were invited to return for the second study. These 40 subjects included 20 COBOL and 20 FORTRAN programmers and were those programmers who had scored in the top and bottom quartiles in the comprehension task in the previous study.

Comprehension results from the second study reinforce and extend the conclusions from the first study, that the understanding of program control flow and procedures precedes understanding of program functions (see Figure 8). This pattern of comprehension results appeared even in the context of a longer, partially documented program after a lengthy study period. Analyses of program summaries also support this conclusion by showing a preponderance of procedural summary statements over data flow and function statements.

****
Insert Figure 8 about here
****

The story of program comprehension does not, however, end with the establishment of a procedural representation. In our second study, a different comprehension pattern emerged after a second exposure to the program during which programmers completed a program modification (see Figure 9). After the modification task, there was a marked shift toward increased comprehension of program function and data flow at the apparent expense of control flow information and this shift was more extreme for programmers who were asked to think aloud while working. This suggests that either the additional time or the goal of modifying the program resulted in a change in the dominant memory representation. The fact that talking aloud while working enhanced this shift suggests that task effects, rather than the extra time alone, are responsible.

****
Insert Figure 9 about here
****

One way to understand this shift in comprehension patterns is to go to theories of text comprehension and speculate about a construct, introduced by van Dijk and Kintsch (1983), that they call a situation model. In this (1983)

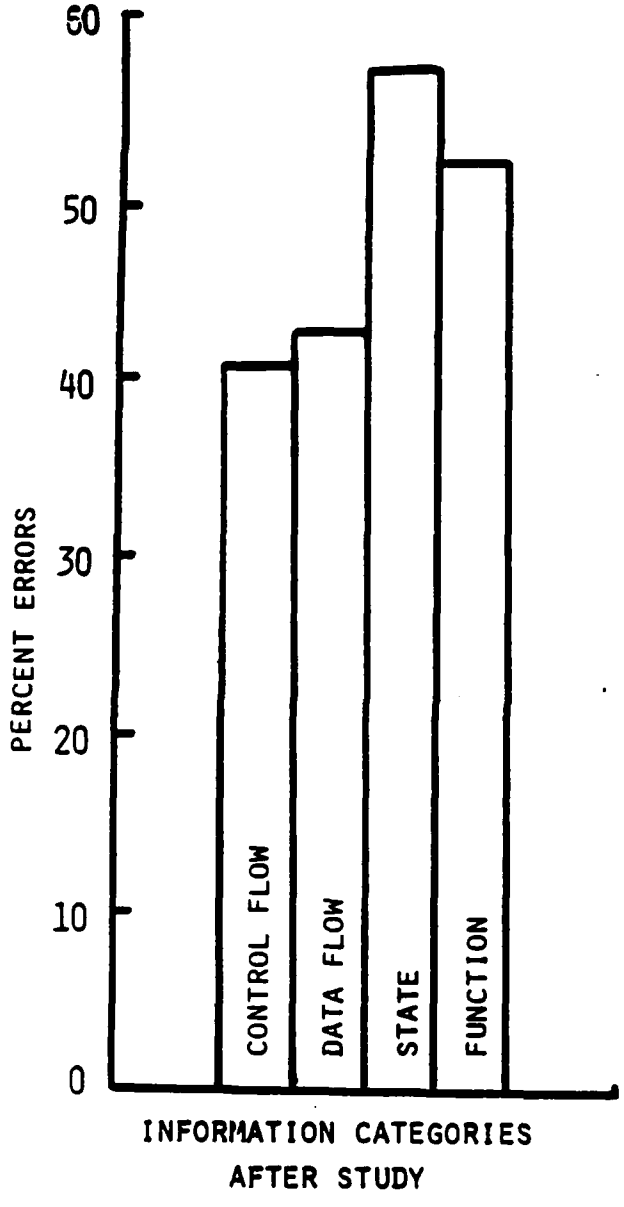


Figure 8. Study Two comprehension question error rates by information category, after Study task.

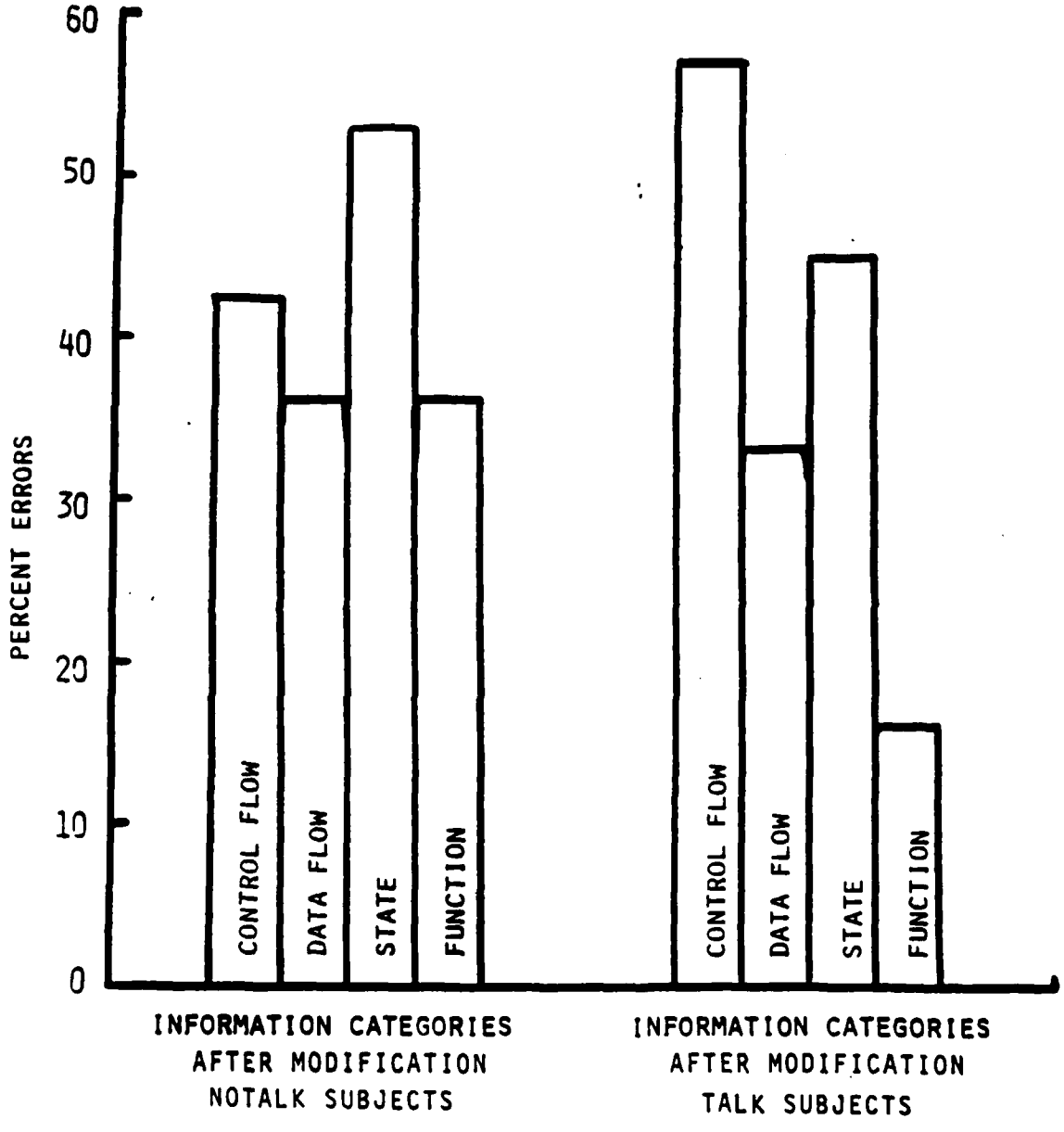


Figure 9. Study Two comprehension question error rates by information category, after Modification task, for Talk and Notalk subjects.

work, van Dijk and Kintsch suggest that two distinct but cross-referenced representations of a text are constructed during comprehension. The first representation, the textbase, includes a hierarchy of representations, consisting of a surface memory of the text, a microstructure of interrelations between text propositions, and a macrostructure that organizes the text representation. The second representation, the situation model is a mental model (e.g., Johnson-Laird, 1983) of what the text is about referentially. In our context, the program text used in the second study is conceptually about searches, merges, computations, and so forth; referentially, it is about cables that take up space, making sure that there is enough space for the cables in the building under design, etc. It is plausible that the functional relations between program procedures are more comprehensible in terms of the real world objects. Thus, the textbase macrostructure may be dominated by procedural relations that largely reflect how programs in traditional languages are structured. The functional hierarchy can be developed with reference to a situation model expressed in terms of the real world objects. Data from our analysis of program summaries in the second study are consistent with this idea: procedural summary statements were most often expressed in terms of program concepts and functional summary statements were most often expressed in terms of the real world object domain.

Van Dijk and Kintsch (1983) also suggest that the construction of the situation model depends on the construction of the textbase in the sense that the textbase defines the actions and events that need explaining. This is consistent with our findings in both studies that procedural representations precede functional representations. In fact, our results suggest that both

time and incentive (talking aloud to an experimenter and having to do a modification) are involved in the successful construction of a functionally based situation model.

A second major purpose of our second study was to descriptively investigate computer program comprehension strategies by analyzing the verbal protocols collected from one-half of the programmers during the program study phase (Pennington, in preparation). We were especially interested in any systematic differences that might appear between the top quartile (Q1) comprehenders and the bottom quartile (Q4) comprehenders, differences that cannot be attributed to experience alone since all of our programmers were highly experienced professionals. Since the top quartile comprehenders showed substantially better comprehension both in our experimental task as well as our more natural task, features of comprehension strategies evidenced in the verbal protocols may well be those that lead to higher levels of comprehension.

As a general summary, we have found that top (Q1) comprehenders are more likely to pursue what we have come to call <u>cross-referencing</u> strategies in comprehension compared to <u>singular</u> strategies more often used by bottom (Q4) comprehenders. We suggested earlier that there are two different "worlds" relevant to a computer program text. One is the "program world" in which various instructions to the computer carry out actions that have effects on values of data objects and the sequence of action execution. The other is the "domain world" in which real world objects exist that are the reason that the program was written. For example, in our second study, the domain world corresponding to the stimulus program was one in which cables were being allocated to locations in a building design. The program world was one in

which lists of numbers were compared against other lists of numbers, some of them added up together and so forth.

When we say that Q4 comprehenders used singular strategies, we mean that they talked about one world or the other almost exclusively. One type of Q4 comprehender followed the program listing in great detail but rarely stopped to coordinate this with why particular program actions were required. The contrasting type of Q4 comprehender used the briefest of clues from the program listing (variable names, a single action) to leap immediately to domain world inferences about what was being accomplished. The latter strategy led to a great many errors concerning the purpose of the program and in a few cases rather fanciful stories about what was going on. The former strategy led to an understanding of detail but later errors in higher level inferences.

When we say that Q1 comprehenders used cross-referencing strategies, we mean that they worked in both worlds, using implications of one world for the other to verify inferences that they were making. For example, after working out a procedure at the program level, they would stop to translate this into the domain world; if the relations in the domain world did not make sense, they would go back to see where they had gone wrong. Conversely, inferences in the domain world would often have implications for what they might expect to see in the program. In these cases, programmers checked in the program to see if their predictions held up; if not, they knew that they didn't have the correspondences right.

Our conclusions regarding strategy differences between Q1 and Q4 comprehenders are supported by analyses of program summary statements and analyses to date of the verbal protocols. We found that Q4 summaries contained

either more detail or more vague (without referent) statements compared to Q1 summaries (Pennington, in preparation). For verbal protocols, Q1 subjects show more transitions between program and domain levels in their inferences and more correct function inferences. These results support the view of program comprehension set forth earlier, that a textbase macrostructure will be dominated by procedural relations reflecting the program world and that a second situation model expressed in terms of the real world domain will be critical for developing a functional hierarchy. Our results also suggest that the mapping between the two worlds and the ability to use one to check the other are central to accurate and complete program comprehension.

REFERENCES

Adelson, B. (1984). When novices surpass experts: The difficulty of a task may increase with expertise. Journal of Experimental Psychology: Learning, Memory, and Cognition, 10, 484-495.

Anderson, J. R. (Ed.) (1981). Cognitive skills and their acquisition. Hillsdale, NJ: Erlbaum.

Boehm, B. W. (1973). Software and its impact: A quantitative assessment. Datamation, 19, 48-59.

Curtis, B., Forman, I., Brooks, R. E., Soloway, E., & Ehrlich, K. (1984). Psychological perspectives for software science. Information Processing and Management, 20, 81-96.

van Dijk, T. A. & Kintsch, W. (1983). Strategies of discourse comprehension New York: Academic Press.

Green, T. R. G. (1980). Programming as a cognitive activity. In H. T. Smith and T. R. G. Green (Eds.), Human interaction with computers. New York: Academic Press.

Green, T. R. G., Sime, M. E., & Fitter, M. J. (1980). The problems the programmer faces. Ergonomics, 23, 893-907.

Johnson-Laird, P. N. (1983). Mental models. Cambridge, MA: Harvard University Press.

Kintsch, W. & van Dijk, T. A. (1978). Toward a model of text comprehension and production. Psychological Review, 85, 363-394.

Newell, A. & Simon, H. A. (1972). Human problem solving. New York: Prentice-Hall.

Pennington, N. (1982). Cognitive components of expertise in computer

programming: A review of the literature. Psychological Documents, 1985, 15, No. 2702.

Pennington, N. & Grabowski, B. (1985). Cognitive components of expertise in computer programming: A conceptual framework. Unpublished manuscript, University of Chicago, Chicago, IL.

Pennington, N. (in press). Stimulus structures and mental representations in expert comprehension of computer programs. Cognitive Psychology.

Pennington, N. (in preparation). Cross-referencing strategies in computer program comprehension. Unpublished manuscript, University of Chicago, Chicago, IL.

Technical Reports and Publications

Pennington, N. (1982, July). Cognitive components of expertise in computer programming: A review of the literature. (Psychological Documents, 1985, 15, No. 2702.)

Pennington, N. and Grabowski, B. (1985, January). Cognitive components of expertise in computer programming: A review and conceptual framework. (ONR Technical Report 1, currently under revision for journal submission.)

Pennington, N. (1985, January). Stimulus structures and mental representations in expert comprehension of computer programs. (ONR Technical Report 2.)

Pennington, N. (1986, September). Stimulus structures and mental representations in expert comprehension of computer programs. (ONR Technical Report 3, also Cognitive Psychology, in press.)

Pennington, N. (in preparation). Cross-referencing strategies in computer program comprehension. (ONR Technical Report 4, in preparation.)

Pennington, N. (1986, December). Expert Programmer Comprehension of Computer Programs: Final Report. (ONR Final Report.)

Acknowledgements and Scientific Personnel

We wish to thank Dr. Henry Halff and Dr. Michael Shafto of the Office of Naval Research for their support and encouragement during the contract period. At the University of Chicago, we received assistance from Beatrice Grabowski, Paul Harvell, John Keating, Helena Szepe, Lori Hunsaker, Jane Ann Layton, and Elizabeth Norman during the course of the project. Colleagues at the Center for Decision Research, Joshua Klayman, Lola Lopes, and Robin Hogarth, provided valuable comments on manuscripts at various stages of completion.

Distribution List [Chicago/Pennington] NR 667-503

Dr. Beth Adelson
Department of Computer Science
Tufts University
Medford, MA 02155

Dr. Robert Ahlers
Code N711
Human Factors Laboratory
Naval Training Systems Center
Orlando, FL 32813

Dr. Ed Aiken
Navy Personnel R&D Center
San Diego, CA 92152-6800

Dr. John R. Anderson
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

Dr. John Black
Teachers College
Columbia University
525 West 121st Street
New York, NY 10027

Dr. Deborah A. Boehm-Davis
Department of Psychology
George Mason University
4400 University Drive
Fairfax, VA 22030

Dr. Jeff Bonar
Learning R&D Center
University of Pittsburgh
Pittsburgh, PA 15260

Commanding Officer
CAPT Lorin W. Brown
NROTC Unit
Illinois Institute of Technology
3300 S. Federal Street
Chicago, IL 60616-3793

Maj. Hugh Burns
AFHRL/IDE
Lowry AFB, CO 80230-5000

Dr. John M. Carroll
IBM Watson Research Center
User Interface Institute
P.O. Box 218
Yorktown Heights, NY 10598

Dr. Fred Chang
Strategic Technology Division
Pacific Bell
2600 Camino Ramon
Rm. 3S-453
San Ramon, CA 94583

Dr. Davida Charney
English Department
Penn State University
University Park, PA 16802

Dr. L. J. Chmura
Computer Science and Systems
Code: 7590
Information Technology Division
Naval Research Laboratory
Washington, DC 20375

Defense Technical
Information Center
Cameron Station, Bldg 5
Alexandria, VA 22314
Attn: TC
(12 Copies)

ERIC Facility-Acquisitions
4833 Rugby Avenue
Bethesda, MD 20014

Dr. Marshall J. Farr
Farr-Sight Co.
2520 North Vernon Street
Arlington, VA 22207

Mr. Wallace Feurzeig
Educational Technology
Bolt Beranek & Newman
10 Moulton St.
Cambridge, MA 02238

Dr. John R. Frederiksen
Bolt Beranek & Newman
50 Moulton Street
Cambridge, MA 02138

Dr. Michael Friendly
Psychology Department
York University
Toronto ONT
CANADA M3J 1P3

Distribution List [Chicago/Pennington] NR 667-503

Dr. Sherrie Gott
AFHRL/MODJ
Brooks AFB, TX 78235

Dr. James G. Greeno
University of California
Berkeley, CA 94720

Dr. Henry M. Halff
Halff Resources, Inc.
4918 33rd Road, North
Arlington, VA 22207

Dr. Bruce Hamill
The Johns Hopkins University
Applied Physics Laboratory
Laurel, MD 20707

Dr. Jim Hollan
Intelligent Systems Group
Institute for
Cognitive Science (C-015)
UCSD
La Jolla, CA 92093

Dr. Ed Hutchins
Intelligent Systems Group
Institute for
Cognitive Science (C-015)
UCSD
La Jolla, CA 92093

Dr. R. J. K. Jacob
Computer Science and Systems
Code: 7590
Information Technology Division
Naval Research Laboratory
Washington, DC 20375

Dr. Robin Jeffries
Hewlett-Packard Laboratories
P.O. Box 10490
Palo Alto, CA 94303-0971

Dr. Wendy Kellogg
IBM T. J. Watson Research Ctr.
P.O. Box 218
Yorktown Heights, NY 10598

Dr. David Kieras
University of Michigan
Technical Communication
College of Engineering
1223 E. Engineering Building
Ann Arbor, MI 48109

Dr. R. W. Lawler
ARI 6 S 10
5001 Eisenhower Avenue
Alexandria, VA 22333-5600

Dr. Clayton Lewis
University of Colorado
Department of Computer Science
Campus Box 430
Boulder, CO 80309

Dr. Stuart Macmillan
FMC Corporation
Central Engineering Labs
1185 Coleman Avenue, Box 580
Santa Clara, CA 95052

Dr. James S. McMichael
Navy Personnel Research
and Development Center
Code 05
San Diego, CA 92152

Dr. James R. Miller
MCC
9430 Research Blvd.
Echelon Building #1, Suite 231
Austin, TX 78759

Dr. Nancy Morris
Search Technology, Inc.
5550-A Peachtree Parkway
Technology Park/Summit
Norcross, GA 30092

Dr. Allen Newell
Department of Psychology
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Distribution List [Chicago/Pennington] NR 667-503

Dr. A. F. Norcio
Computer Science and Systems
Code: 7590
Information Technology Division
Naval Research Laboratory
Washington, DC 20375

Dr. Donald A. Norman
Institute for Cognitive
Science C-015
University of California, San Diego
La Jolla, California 92093

Dr. Judith Reitman Olson
School of Business
Administration
University of Michigan
Ann Arbor, MI 48106

Dr. Peter Polson
University of Colorado
Department of Psychology
Boulder, CO 80309

Dr. Steven E. Poltrock
MCC,
Human Interface Program
3500 West Balcones Center Dr.
Austin, TX 78759

Dr. William B. Rouse
Search Technology, Inc.
5550-A Peachtree Parkway
Technology Park/Summit
Norcross, GA 30092

Dr. Marc Sebrechts
Department of Psychology
Wesleyan University
Middletown, CT 06475

Dr. Colleen M. Seifert
Intelligent Systems Group
Institute for
Cognitive Science (C-015)
UCSD
La Jolla, CA 92093

Dr. Sylvia A. S. Shafto
Department of
Computer Science
Towson State University
Towson, MD 21204

Dr. Elliot Soloway
Yale University
Computer Science Department
P.O. Box 2158
New Haven, CT 06520

Dr. Ralph Wachter
JHU-APL
Johns Hopkins Road
Laurel, MD 20707

Dr. Barbara White
Bolt Beranek & Newman, Inc.
10 Moulton Street
Cambridge, MA 02238

Dr. Wallace Wulfeck, III
Navy Personnel R&D Center
San Diego, CA 92152-6800

Dr. Joseph L. Young
Memory & Cognitive
Processes
National Science Foundation
Washington, DC 20550

END

5-87

DTIC